MPICH on the T3D: A Case Study of High Performance Message Passing [*]

Ron Brightwell [†]
Anthony Skjellum
Integrated Concurrent and Distributed Computation Research Lab and
NSF Engineering Research Center
Mississippi State University
bright@cs.sandia.gov
tony@cs.msstate.edu

Abstract

This paper describes the design, implementation and performance of a port of the Argonne National Laboratory/Mississippi State University MPICH implementation of the Message Passing Interface standard to the Cray T3D massively parallel processing system. A description of the factors influencing the design and the various stages of implementation are presented. Performance results revealing superior bandwidth and comparable latency as compared to other full message passing systems on the T3D are shown. Further planned improvements and optimizations, including an analysis of a port to the T3E, are mentioned.

1. Introduction

As part of the MPICH project between Argonne National Laboratories and Mississippi State University, a port of the MPICH implementation of MPI to the T3D was designed and implemented. This was not a rote exercise, but rather an in-depth effort that stressed the internal abstract device interface design of MPICH, demonstrated high performance, while revealing several interesting issues concerning MPI on systems that have distributed shared memory primitives at a low level. While there were a number of bugs in this implementation early on (including some incidental to MPICH), the experiences associated with removing these bugs and retaining high performance are illuminating. Furthermore, the first-principles MPICH device created for this port stands apart from others written using the channel interface or P4, and so is an important contribution to the overall device set of MPICH. Finally, issues concerning misalignment and datatype-dependent performance have been identified and should be factored into emerging MPI test suites.

The highest bandwidth of full message-passing systems on the T3D has been achieved, with just a little help from Cray, outstripping the PVM and EPCC MPI implementations. However, it is clear that vendor support would have been helpful, inasmuch as the other message passing systems, which have vendor sanction, have also had access to support for removing the subtle bugs that arise in pushing the envelope of performance and functionality in the T3D runtime environment. Higher bandwidth has been achieved on collective operations, though they are far from optimal.

1.1. MPICH

MPICH is a portable implementation of the Message Passing Interface (MPI) [3] standard developed jointly by Argonne National Laboratory and Mississippi State University. MPICH contains an abstract device interface (ADI) upon which a high-level message passing application programmer interface such as MPI can be implemented. The ADI performs four main functions [9]:

- Sending and receiving
- Data transfer
- Queueing
- Device-dependent functions.

Porting MPICH to an architecture such as the T3D involves the creation of new "device" that interacts with the ADI through a set of routines (see [8] for details) and handles. These handles are used to cache device specific data to pass information between the device independent and device dependent layers of MPICH.

[*] Work supported in part by the Department of Energy
[†] Presently on staff at Sandia National Laboratories, Albuquerque, NM

1.2. The Cray T3D

The Cray T3D is a massively parallel system which contains up to 2048 processors connected by a high-speed, 3-D torus communication network [6]. Cray T3D has a physically distributed shared memory, where each processing element (PE) has local memory which is globally addressable. The T3D model is one process per PE and any PE can directly address any word of memory on any other PE. Cache consistency is the resonsibility of the user.

2. Design Decisions

The initial design decision was to choose the most efficient method of communication between processing elements. Cray's Block Transfer Engine (BLT) was an original consideration, but this method had some major drawbacks. Using the BLT requires the overhead of making an expensive system call. The asynchronous capability of the BLT was appealing, but because of limited memory bandwidth, BLT was taken out of consideration. Each processing element shares a BLT engine with another processing element. When the BLT is in use by one PE, the second PE will block if it tries to access the BLT. The BLT also requires a flush of the entire cache upon transfer.

Cray also offers a direct shared memory access library (SHMEM) [7] for remote memory transfers. This library contains a plethora of functions for point-to-point and collective communication, synchronization, and cache manipulation. The two basic operations in this library are *shmem_get()*, which copies data from a remote PE to the local PE, and *shmem_put()*, which copies data from the local PE to the remote PE. Multiples of 32- and 64-bit transfers are supported, with aligned data.

After investigating both methods, and on the informal advice of Cray Research [11], the shared memory library was chosen as the means of communication upon which to build the MPICH T3D device. Further investigation into the shared memory library revealed that *shmem_put()* transfers data at nearly twice the bandwidth of *shmem_get()*. Therefore, *shmem_put()* was chosen as the basis for the implementation of the device.

In order to use *shmem_put()* to transfer data, a remote address on the receiver must be known *a priori*. Therefore, the next step in the design process was determining a method by which a sender could obtain a target address to which a message could be sent. This method must also maintain pairwise ordering for messages (within a communicator) in order to be MPI compliant [3].

Several possibilities were considered, based on the available shared memory constructs and functions in the SHMEM library. The *shmem_swap()* function provides an atomic swap operation, and was originally considered as a means to gain atomic access to a pre-allocated message buffer at the receiver. However, the latency cost associated with such an operation was thought to be too high.

Global and static variables and dynamic memory allocated with the *shemalloc()* function are guaranteed to have the same address on every PE. It was decided that a message buffer could just be an offset into a array of message buffers allocated from the shared heap. In fact, remote memory writes to global or shared locations is the method encouraged by SHMEM documentation [5, 7]. In order to maintain pairwise ordering of messages, a sender was to have only one buffer into which messages would be written at the receiver, and at any time there could only be one outstanding message between a sender and a receiver.

Once the low level communication design was complete, a design for efficient implementation of MPI communications was developed. A two-level protocol for sending messages was decided upon: a protocol for short messages in which header information for each message would accompany the body of the message, and a protocol for longer messages in which the body could be delivered directly into the user buffer at the receiver. These protocols would allow for the smaller messages to be sent quickly and also allow for a limited global buffer space necessary for put-based communication. An additional protocol layer would be to added to handle synchronous communications.

3. Implementation

None of the then-existing devices in the MPICH implementation used a put-based shared memory strategy for communications. Therefore, a completely new device was created for the Cray T3D, rather than building upon or augmenting an existing device. MPICH's ADI provided the required functions for message queueing, so the new device was responsible for sending and receiving messages, transfer of data to the API, and a few other device specific functions.

3.1. Sending and Receiving Messages

For the MPICH implementation, each process in the application allocates an array from the shared heap that contains a slot into which every other process (including itself) can send messages (receive buffers) and an array of flags for each of its buffers on every other process (send flags) (Figure 1). Allocating from the shared heap insures that both of these data structures reside at the same address on every process. The send flags indicate to the sender the state of its receive buffer at the receiver, and is a method of flow control so that successive messages to the same receiver are not overwritten and remain pairwise ordered.

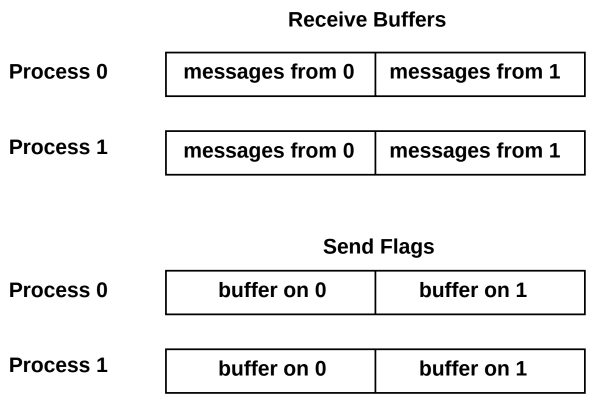

Figure 1. Receive Buffers and Send Flags.

Before any message can be sent, the sender must wait for the send flag associated with the receive buffer on the receiving PE to be clear. This busy waiting involves traversing its own receive buffers looking for incoming messages to process so that communications may progress. As soon as the send flag is clear, the sender sets the send flag and the outgoing message is written to the receiver at the sender's message slot. A new message at the receiver is signaled by a status flag contained in each receive buffer. This status flag is set when the sender writes a message header into its receive buffer.

Traversal of the receive buffers by the receiving PE is implemented as fairly as possible, with the search beginning at the first buffer beyond where the last message was received. Upon discovering a new message, the receiving PE processes the message, clears the receive buffer flag, and then informs the sending PE that its receive buffer is free by clearing the send flag at the sending PE.

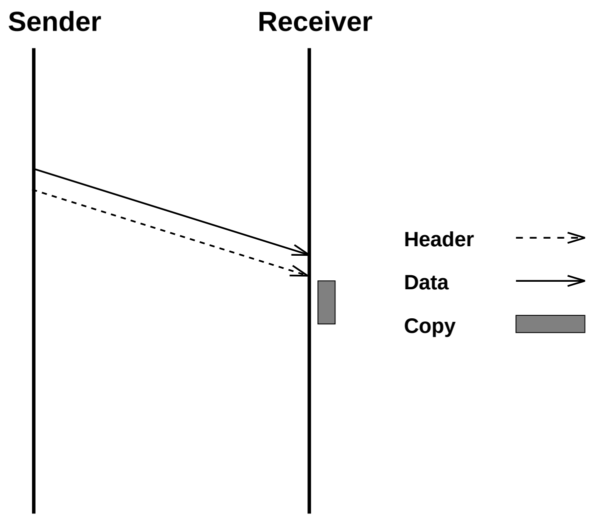

Figure 2. Short Send Protocol.

For messages using the short protocol (Figure 2), both a message header and the user data are sent. The message header contains the following information:

- mode value identifying the type of message (short regular, long synchronous, etc.)
- context id of the communicator being used
- local rank of the sender within the communicator
- message tag
- message length
- status flag indicating the buffer is in use

The data is written to the receiver before the header is written, insuring that the data will be valid when the receiver recognizes that the buffer contains a new message. If the data is not four-byte aligned, it is copied to the sender's own locally aligned receive buffer before it is written to the receiver. The only use for the sender's own local receive buffer is as a copy space.

Upon discovering any new message, the receiver searches its posted receive queue against the context id, local rank, and tag values for a match. If the search is successful, the data is copied from the receive buffer into the application's designated buffer. Both sending and receiving of this message is complete. If the search is unsuccessful, space for the data is allocated and the data is copied from the receive buffer into the newly allocated buffer. This message is then added to an unexpected message queue. Only the sending side of this message is complete.

The receiver then clears the status flag in this receive buffer and informs the sender that its receive buffer is now free by writing a cleared status value into its designated send flag at the sender.

For messages using the long protocol (Figure 3), only a header is sent. The message header for the long protocol is identical to the short message header, plus the the following additional information:

- local address of a structure where the receiver can write (using a put) the following information:
  - location of the receive buffer
  - location of the receiver's completed flag
  - length of the receive buffer

The sender initializes the buffer length field to a negative value, and writes only this header to the receiver. Upon discovering this new message, the receiver again searches the posted receive queue for a match, allocating a buffer if unsuccessful (an unexpected message). For the long protocol, a structure containing the address of the buffer, the length

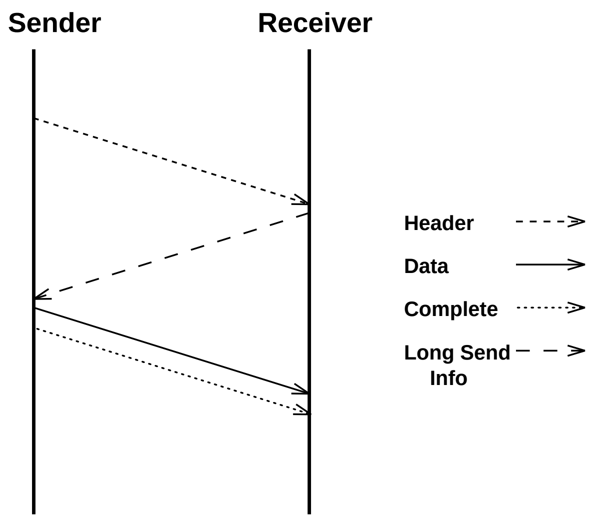

Figure 3. Long Send Protocol.

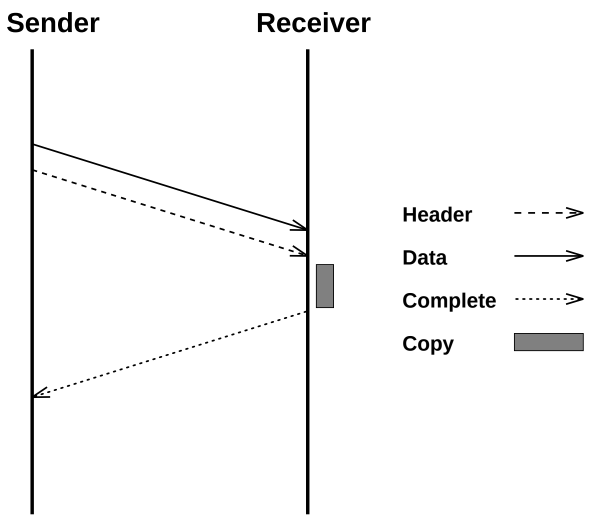

Figure 4. Short Synchronous Send Protocol.

of the buffer, and the address of the receiver's request completed flag is filled in and written back to the sender at the location specified in the message header.

Once this header has been processed by the receiver, the reciever clears the status flags of the receiver buffer and send flag, just as in the short protocol.

At the sender, a non-negative receive buffer length signals that the receiver has processed the message header and the user data may be written to the receiver at the specified location. After the user data is written, the sender writes a completed flag value to the receiver's completed flag location, informing the receiver that the data has been written. If this message was expected at the receiver, both sending and receiving of this message is completed. If it was unexpected, only the send operation is complete, and the message is added to the queue of unexpected messages at the receiver.

For synchronous send operations, both the short and long message headers contain the following additional information:

- local address of the send completed flag where the receiver can write a completed flag

For both long and short protocol messages, when the receiver recognizes the completion of a synchronous receive operation, a completed flag is written to the location at the sender specified in the message header (Figure 4 and Figure 5). Completion of a synchronous send operation is not complete at the sender or the receiver until the receiver updates the send completed flag.

When a receive is posted, the unexpected message queue is searched. If the search is successful, an unexpected message handle is associated with the posted receive handle.

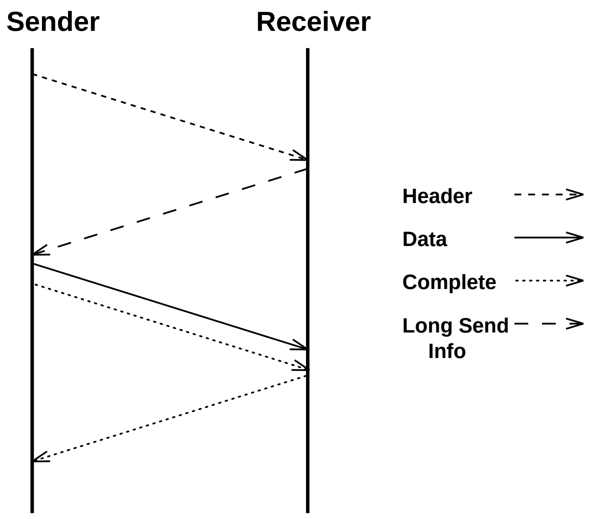

Figure 5. Long Synchronous Send Protocol.

Once the unexpected message is completed, the data is copied from the allocated buffer to the application specified buffer. If the search is unsuccessful, the receive handle is added to a queue of posted receives. This queue is searched every time a new message is discovered in the receive buffers.

This send protocol prohibits taking advantage of any opportunities for optimization provided by the MPI ready send functions. Therefore, ready sends are equivalent to blocking sends.

3.2. Cache coherency

Because remote memory updates take place without the involvement of the remote processor, the cache on the remote PE can become invalid. The SHMEM library provides several functions used to help ensure cache coherency. Our initial implementation chose the simplest of these methods. Automatic cache invalidation for all writes into local memory by other PE's can be enabled by a call to *shmem_set__cache_inv()*. This method was chosen rather than invalidating individual cache lines or flushing the entire data cache whenever a receive is posted.

3.3. Address Validation

Extreme caution must be taken when using *shmem_put()* to write into a non-global address on a remote PE. Global variables, static variables and memory allocated from the global shared heap using *shmalloc()* are guaranteed to be identical and valid on every PE. However, automatic variables allocated from the local stack and dynamic memory allocated from the local heap are not guaranteed to be valid on every process. The *shmem_put()* function checks the validity of both the source *and* target addresses in the local process' address space. Should the target address not be a valid address in the sender's address space, an operand range error is generated, and, if not caught, results in the application dumping a corefile. In order to write to any address on another PE, the target address must be made valid at the sender.

Cray Research (CRI) pointed out an undocumented function, *malloc_brk()*, which exists for validating memory allocated from the local heap using *malloc()*. *malloc_brk()* essentially works like the *brk()* system call, exapanding the heap as necessary to make the target heap address valid. However, unlike *brk()*, the extra memory is added to the *malloc()* free list for use by the application. CRI also contributed an assembly routine, *shmem_stack()*, for validating memory allocated from the local stack. *shmem_stack()* extends the local stack if the target address is beyond the top of the local stack.

There are only a few places in the implementation where the target address must be checked for validity and be made valid. In the long protocol, the receiver must validate the address at the sender where the structure containing the location of the receive buffer, the location of the receiver's completed flag, and the length of the receive buffer are written. Likewise, the sender must then validate this receive buffer location before the user data can be written and also the receive completed location before the receive completed flag can be written. Similarly, in the synchronous protocol, the receiver must validate the location of the send completed flag at the sender before the flag can be updated. All other puts are done to memory allocated from the shared heap.

Checking for an invalid address is done by comparing the target address with both the top of the stack and also with the current break value obtained from *sbrk()*. Since there was no accurate means by which to get the value of the top of the stack, a simple assembly routine returning the stack pointer was written. Should the target address be less than the top of the stack or greater than the current heap break value, the target's distance from each of those two limits is calculated. If the target is closest to the top of the stack, the stack is expanded, and if the target is closest to the heap break value, the heap is extended. The costs associated with checking the validity of a target address and extending the stack are nominal, but extending the heap involves making system calls to *malloc_brk()* and *sbrk()*.

3.4. Alignment

Because *shmem_put()* can only transfer data that is four-byte aligned, temporary buffers are allocated for transfers involving addresses which are either not four-byte aligned or which are not a multiple of four in length. A temporary send buffer is allocated for a long protocol send that originates from an address that is not four-byte aligned. A temporary receive buffer is also allocated in the long protocol for a receive that is destined for a buffer which is not four-byte aligned or whose length is not a multiple of four. Consequently, sending and receiving to and from misalinged buffers has a substantial performance degradation that could be improved by a more optimal implementation. However, since character data is the only type which is not four-byte aligned, and due to the associated additional code complexity, efforts toward optimization of misaligned buffer use were considered to be of low priority.

4. Performance

Performance tests were run using the mpptest program contained in the MPICH distribution. The tests were run with the default parameters, using only the '-size' switch to modify the start, end, and increment message sizes. The tests compare the current MPICH implementation with Cray Research/Endiburgh Parallel Computing Centre implementation version 1.4a. All tests were run on two processors.

Figure 6 compares the latency for message lengths from zero to 1024 in increments of 32 bytes. While the MPICH numbers are erratic, the performance is comparable to that of the CRI/EPCC.

Figure 7 compares the bandwidth for message lengths from 10k to 200k in increments of 10k. Bandwidth of the CRI/EPCC version levels off to around 29 megabytes per second starting at messages of 100k. However, the MPICH bandwidth continues to increase, leveling off to approximately 100 megabytes per second at messages of 100k. Figure 8 shows the continuation for message sizes from one

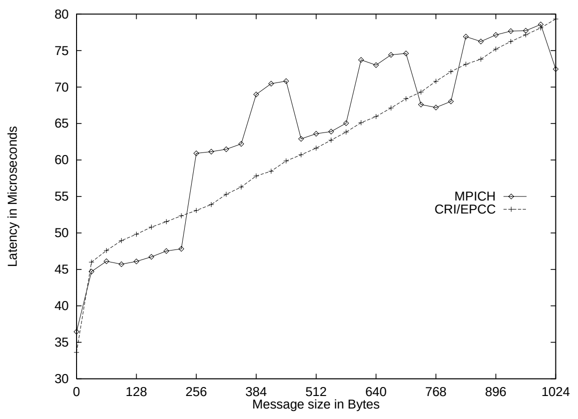

Figure 6. Small Message Latency.

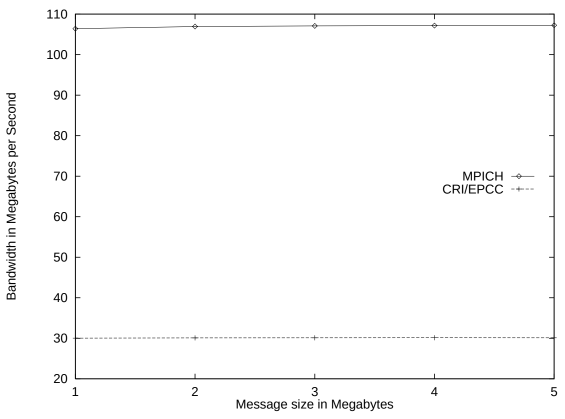

Figure 8. Large Message Bandwidth.

to five megabytes in the length. The MPICH implementation levels off at around 107 megabytes per second, achieving approximately 85% of the available peak bandwidth, while the CRI/EPCC version continues to hover around 30 megabytes per second.

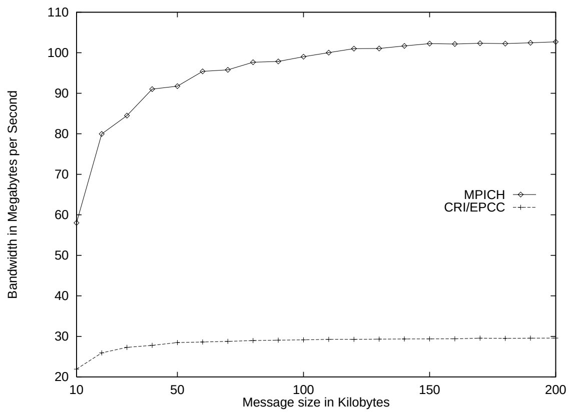

Figure 7. Medium Message Bandwidth.

5. Stages of Implementation

The original device for the T3D was built from an existing device constructed for the Myrinet gigabit network [1]. Even though a device for shared memory communications existed in the MPICH distribution, the code was judged to be too complex to either integrate a strictly put-based shared memory implementation into or to use as a starting point for such a device. The complexity of the code for the device for Myrinet was much less, and the learning curve associated with implementing a T3D device from the Myrinet device was judged to be much lower. Only the basic framework of the device was retained and all code and device dependent structures were eliminated.

As a result, implementing the first T3D device required only approximately one month. However, because the device for Myrinet was packet-based and was not designed for shared-memory-type operations, the T3D device had to be shaped into one which was. Subsequent improvements made over the course of four months worked to optimize the device for a distributed shared memory environment.

The initial device was crude and did not properly handle message buffers that were not eight-byte aligned. An initial improvement involved replacing *shmem_put()* with *shmem_put32()*, the SHMEM function for transferring four-byte aligned quantities. However, buffers that were not four-byte aligned or a multiple of four in length were still not properly managed.

Further improvements to the device fixed bugs associated with the long send protocol. In the initial implementation of the long send protocol, both blocking and non-blocking sends were handled identically. After sending a long message header to the receiver, the sender would enter a busy wait loop in the device layer waiting for the receiver to respond. This method did not take advantage of the opportunities for increased performance offered by non-blocking send operations. The implementation was modified so that the long send protocol would simply write a long send header to the receiver when the send was posted and would try to complete the send at some later point in time. This implemenation caused protocol failures for certain combinations of wait and test operations. The current device contains a queue of incomplete long send request handles. When testing or waiting on a receive request handle, the long send request must be traversed so that long sends make progress [2].

The biggest problem with the MPICH T3D device was its propensity for spurious message loss. This because of a misunderstanding of how *shmem_put()* messages were received. In the first implementation stages, the status flag that notified the receiver of a new message was the first field in the message header structure. Even though the header was written in one ‘message’, the status field could be in a different cache line than the rest of the header information. A receiver that was traversing its receive buffers looking for status flags to be set could possibly recognize a set status flag and copy the other header information before it was valid or even written to memory. As such, the message would be received, but it most likely would contain incorrect values in the context or tag fields and end up in the unexpected message queue. This problem was solved by moving the status flag so that it is the last value written into the receive buffer, insuring that all other header information is valid when a set status flag is discovered.

CRI introduced a bug by changing the implementation of *malloc_brk()* so that the *shmem_stack()* routine was extending the stack to an illegal value. The symptoms of this problem were recognized without the help of CRI. Test codes exhibited operand range errors upon entering functions subsequent to a call to *shmem_stack()* to validate a target address. This problem was fixed by saving the stack pointer before the *shmem_stack()* call and resetting the stack pointer afer the *shmem_put()* call. An assembly routing was written to reset the value of the stack pointer. The cause of this problem was only surmised, and while this seemed to be the only solution, recent information from CRI confirmed our suspicions and the validity of the solution.

6. Future Work

There are many performance improvements and enhancements that need to be investigated for this device. The current implementation only transfers contiguous blocks of data, packing and unpacking non-contiguous datatypes when needed. Use of strided puts with the *shmem_ixput()* function for indexed data types, or even multiple puts for vector datatypes needs to be studied.

The design of the send flags and receive buffers provides the ability to do accomplish control so that buffering unexpected messages may be turned on or off or even configured to use only a set amount of memory. This desirable feature has not currently been utilized as a means of reducing the amount of required buffer space.

The collective operations are the default MPICH collective operations which are built on top of MPI point-to-point communications. While these have shown good performance on the T3D, building MPI collective communications on top of the SHMEM collective operations needs to be investigated promptly.

Work is ongoing on the next generation ADI [10]. The goal of this new ADI is to eliminate as much overhead as possible and achieve lower latencies than the first generation ADI for common cases, such as sending and receiving contiguous datatypes. Additionally, the new ADI should maintain ease of implementation and retain opportunities to take advantage of the advanced capabilities of the underlying hardware.

The MPICH T3D device can and should also be used as a basis for a port to the T3D’s successor, the T3E [4]. The shared memory library on the T3E has elminated much of the complexity of the T3D device by augmenting functionality. T3E systems have automatic cache coherency, so the device need not explicitly invalidate the cache on remote memory writes. The T3E also does not attempt to validate remote addresses on the local PE, correcting the T3D flaw. The checking and validating of remote addresses using *shmem_stack()* and *malloc_brk()* will not be required. A major difference between the T3D and T3E will be the ability to have out-of-order puts because of adaptive 3D routing. Currently, the T3D ensures that successive puts to the same PE will arrive in the order sent. On the T3E, this may not be assumed. A library function, *shmem_fence()*, must be called between successive puts to insure that the puts will occur in the order issued. This adaptive routing feature can possibly be exploited for collective as well as point-to-point communications.

7. Acknowledgments

Gratitude is expressed to Peter Rigsbee of CRI for guidance in choosing the best communication facilites and for information regarding *malloc_brk()* and *shmem_stack()*. Thanks also to Karl Feind of CRI for information about the T3E. Gratitude is also expressed to Shane Hebert of Mississippi State University for his work testing the various implementations. And finally, deep thanks are extended to Rusty Lusk and Bill Gropp of Argonne National Laboratory for their help in this endeavor.

References

[1] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Miyrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.

[2] G. Burns and R. Daoud. Robust MPI message delivery with guaranteed resources. MPI Developers Conference, June 1995.

[3] L. Clark, I. Glendinning, and R. Hempel. The MPI Message Passing Interface Standard. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1994.

[4] I. Cray Research. The Cray T3E series. http:// www.cray.com/ PUBLIC/product-info/T3E/overview.html.

[5] Cray Research, Inc. *Cray Research MPP Software Guide, SG-2508 1.1*, 1994.

[6] Cray Research, Inc. *Cray T3D System Architecture Overview, HR-04033*, March 1994.

[7] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3,* October 1994.

[8] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*. Mathematics and Computer Science Division, Argonne National Laboratory, October 1994.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[10] W. Gropp and R. Lusk. MPICH working note: The second-generation ADI for the MPICH implementation of MPI. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, February 1996.

[11] P. Rigsbee. Personal correspondence, January 1995.